Implementing Kilo-Instruction Multiprocessors

Enrique Vallejo[1], Marco Galluzzi[2], Adrián Cristal[2], Fernando Vallejo[1], Ramón Beivide[1], Per Stenström[3], James E. Smith[4] and Mateo Valero[2]

[1]*Grupo de Arquitectura de Computadores, Universidad de Cantabria*
[2]*Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya*
[3]*Dept. of Computer Science and Engineering, Chalmers University of Technology*
[4]*Dept. of Electrical and Computer Engineering, University of Wisconsin-Madison*

Abstract

Multiprocessors are coming into wide-spread use in many application areas, yet there are a number of challenges to achieving a good tradeoff between complexity and performance. For example, while implementing memory coherence and consistency is essential for correctness, efficient implementation of critical sections and synchronization points is desirable for performance.

The multi-checkpointing mechanisms of Kilo-Instruction Processors can be leveraged to achieve good complexity-effective multiprocessor designs. We describe how to implement a Kilo-Instruction Multiprocessor that transparently, i.e. without any software support, uses transaction-based memory updates. Our model not only simplifies memory coherence and consistency hardware, but at the same time, it provides the potential for implementing high performance speculative mechanisms for commonly occurring synchronization constructs.

1. Introduction

Multiprocessors are rapidly becoming the standard for computing platforms, including simple single-board and even single-chip systems. Many of these designs implement shared memory multiprocessors because shared memory provides a clear programming model where sharing code and data structures is simplified. However, there are several aspects that complicate the design and programming of a shared memory system and limit its performance. Among these complicating, potentially restrictive aspects are:

- The *interconnection hardware*, which provides quick access to data held in remote memory -- Accessing data from remote memories is much slower than from local memory, increasing the effective memory latency. This latency can be increased even more by the overhead that a coherence protocol imposes.
- The *coherence protocol*, which manages the correct sharing of memory values among private caches -- Basic coherence protocols are based on a memory directory or a broadcast bus, although significant work has been done to simplify protocols and improve performance [19][15].
- The *consistency model*, which manages the correct ordering of memory operations to different memory locations -- Sequential Consistency (SC) [17] is the most desirable model as it provides the most intuitive programming model, but it most often requires that memory operations from each program appear to be executed in-order which may limit the system performance. Other consistency models achieve a higher performance by relaxing these constraints [25], at the cost of a more complex programming framework.
- The need for data sharing can generate access conflicts, which are solved with *exclusion mechanisms* -- Locks ensure sequential access to shared data by stalling other processors that try to access the same critical section simultaneously. Sometimes critical sections are created in a conservative manner which can degrade performance. Previous works [21][27][26][14][29] have proposed the speculative execution of critical sections, improving performance in case there is no real data contention.
- Finally, *synchronization operations* which ensure that different program threads can cooperate with each other -- These operations, such as barriers or flags, can also stall processors when a synchronization wait is needed. Some proposals [21][14] also deal with this problem by using speculative execution.

A way to solve the above-mentioned problems is to employ **transactional memory systems**. These systems implement memory operations as transactions that are guaranteed to be atomic, and which simplify

0-7803-9032-6/05/$20.00 ©2005 IEEE

the design or improve the performance of a multiprocessor system. Much work has been done on this field and we list some important recent work:

- *Transactional Lock Removal* (TLR) [27] is a method that detects critical sections, and dynamically substitutes them with transactions, eliding the lock acquisition. Transactions are used as a substitute for critical sections, with the aim of augmenting parallelism when no access conflict occurs.
- *Thread-level Transactional Memory* (TTM) [23] is a software-hardware approach that covers different levels of the system. They implement transactional support at thread level, so that the programmer specifies the start and finish of transactions, with support in lower levels of the operating system and hardware.
- *Transactional Coherence and Consistency* (TCC) [14] proposes a new shared memory model where atomic transactions are the basic unit for communication, simplifying both parallel applications and coherence and consistency hardware. Similar to TTM, this proposal modifies the programming model, forcing the programmer to divide the program into transactions.

In prior work, the memory latency problem has been shown to be significantly reduced by **Kilo-Instruction Processors** [4][6][7]. These processors can hide the memory latency by supporting thousands of in-flight instructions. Among the different mechanisms proposed to enable this very large number of in-flight instructions, the primary one is multiple checkpointing [2] which easily facilitates speculative execution.

Later, in [10] the idea of having a multiprocessor composed of Kilo-Instruction Processors, which is intuitively called **Kilo-Instruction Multiprocessor**, is introduced for the first time. This work just evaluates the performance potential of the new system.

In this paper, we describe a *correct implementation* of a Kilo-Instruction Multiprocessor, which will henceforth be referred as **KIMP**, which takes advantage of the underlying uniprocessor mechanisms to simplify the design and to improve performance. This way, KIMP leverages the multiple checkpointing mechanism of the Kilo-Instruction Processors 1) to perform memory updates in a transactional manner, and 2) to apply speculation mechanisms to execute through critical sections and synchronization points in a straightforward manner.

Performing memory updates in KIMP as if they are transactions leads to *implicit transactions*; i.e., they are orchestrated in hardware, without any software support and are transparent to the programmer. Consequently, the proposed KIMP comprises all the advantages of a transactional memory system, including the implementation of sequential consistency, in a natural and efficient manner. On the other hand, having the ability to speculate through critical sections and synchronization points can improve system performance when such constructs are conservatively coded. Hence, the KIMP design is a high-quality complexity-effective option for future multiprocessor systems.

The remainder of the paper proceeds as follows. In Section 2, an overview of the mechanisms and the expected performance of the Kilo-Instruction Processors is given, in single and in multiprocessor systems respectively. Section 3 introduces KIMP, with an initial overview. Section 4 explores the implicit transactional behavior of KIMP systems, and some of the advantages that it provides. Sections 5 and 6 explain the details about memory coherence and consistency in KIMP. Sections 7 and 8 provide some ideas to improve performance by speculating past locks and barriers. Finally, some conclusions are given in Section 9.

2. Background

2.1. Kilo-Instruction Processors

The first work on single Kilo-Instruction Processors [2] demonstrated in detail their ability for hiding large latencies, specifically due to memory accesses, because the processor allows thousands of instructions to be in-flight at the same time. However, in order to increase the number of in-flight instructions we must increase the capacities of several resources, the most important ones being the re-order buffer or ROB, the instruction queues, the load/store queues and the physical registers. Unfortunately, simply up-sizing these structures, is not feasible with current and near-term technology.

In order to overcome the difficulties of up-sizing critical structures, Kilo-instruction processors employ a number of different techniques, to arrive at an overall implementation. Such an approach is possible because critical resources are underutilized in present out-of-order processors as has been shown in [3][16].

The main technique consists of multi-checkpointing long latency instructions instead of having a large ROB [2]. This way, instructions can be committed in an out-of-order fashion, with the possibility of freeing more resources than in normal processors. The second technique is the Slow Line Instruction Queue that proposes a secondary instruction queue to which long-latency instructions can be moved [5]. This mechanism

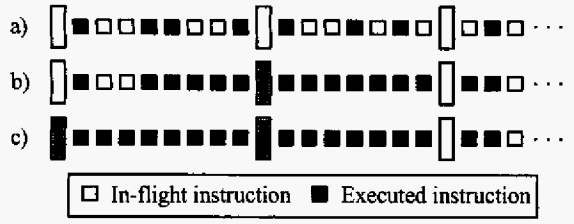

Figure 1: Multiple checkpoints.

allows the regular instruction queue to remain small and fast. The last technique is called Ephemeral Registers. It is an aggressive register recycling mechanism [20], which combines delayed register allocation and early register recycling and, in conjunction with multicheckpointing and Virtual Tags [22], it allows the processor to non-conservatively de-allocate resources.

Multi-checkpointing. At specific instructions during program execution, generally branches and long latency instructions like loads that miss in the L2 cache, a checkpoint is taken. The checkpoint is a snapshot of the processor state. If an exception or branch misprediction occurs, instead of using a ROB and flushing up to the excepting instruction, the state is rolled back to the closest checkpoint prior to the excepting instruction, leading to a longer recovery time. However, to minimize the misprediction penalty a reduced structure called a pseudo-ROB is additionally used [5]. The pseudo-ROB only maintains the youngest in-flight instructions and allows precise recovery of these instructions in a manner similar to a conventional ROB. Because exceptions and branch mispredictions fall most frequently within these youngest instructions, the average recovery time is effectively reduced. Therefore, using a relatively small set of checkpoints for long flight time instructions assures safe points of return and reduces ROB requirements considerably.

Given that multiple checkpoints are taken during program execution, the mechanism works as follows. As the pipeline advances, in-flight instructions corresponding to the different checkpoints are executed. When these speculatively executed instructions finish, they remain in the processor queues if they are memory operations, otherwise are flushed from the processor, leaving their results to the corresponding registers. Only when all the instructions corresponding to the oldest checkpoint are finished, the processor commits the checkpoint atomically, consequently removing memory operations from the load or store queue and committing all the results speculatively calculated. Then, with this action, the speculative instructions become globally performed. Figure 1 shows an example of the multiple checkpointing mechanism, where oldest instructions are to the left. Black instructions are finished, but their results are not committed. In case a), instructions from three consecutive checkpoints are in flight. In case b), the second checkpoint is finished, as all the instructions within it are finished, but it can not commit as it is not the oldest checkpoint in the pipeline. In case c), the first checkpoint (the oldest one) finishes, so it and the second checkpoint can commit, making the third checkpoint to be the new oldest checkpoint.

2.2. Kilo-Instruction Multiprocessors

This previous work by A. Cristal et al. and the problem of increased latencies in multiprocessors are the motivation for Kilo-Instruction Multiprocessors. In this first approach, we use a number of Kilo-Instruction Processors to construct a small-scale non-uniform memory access (NUMA) multiprocessor. The new multiprocessor configuration, firstly published in [10], has been shown to effectively hide large latencies coming from both local and remote memory accesses, including latencies due to the interconnection network. Figure 2 shows the reduction in the execution time for different benchmarks from the Splash2 suite, when using 1024 in-flight instructions as compared with 64; the memory access time is 500 cycles.

This work explores for the first time multiprocessors based on processors that use checkpointing as the basic mechanism for improving performance, while other previous work like [30] make use of checkpointing just for fault tolerance. However, in [10] only an evaluation of the performance achieved by the system is done, leaving out any kind of architectural detail. Describing the system architecture in more detail is one of the goals of this paper, i.e., we show one possible design for checkpoint-based multiprocessors.

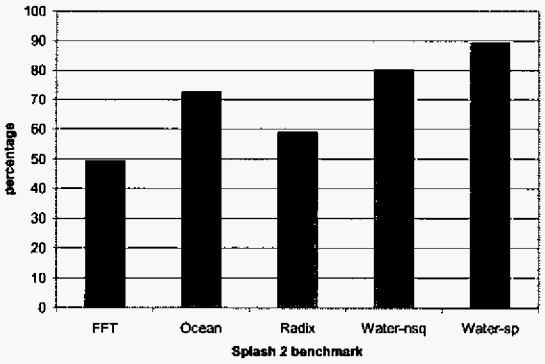

Figure 2: Performance potential of Kilo-Instruction Multiprocessors.

3. KIMP overview

The KIMP design is based on a shared memory multiprocessor. In particular, we propose as the basic system a small-scale multiprocessor where the nodes are Kilo-Instruction Processors interconnected via a snoopy bus. For the sake of simplicity, we present an SMP architecture, although the idea can be applied to CMP or DSM systems. Our proposal takes advantage of the main mechanism of the Kilo-Instruction Processors, that is, the multiple checkpointing, so the cost of implementing a KIMP can be very low.

Figure 3 shows an example of the execution flow from four processors, P_1 to P_4, and their respective checkpoints. Different processors execute different portions of code, taking different checkpoints as the execution advances, and being able to roll back execution to a certain checkpoint in case of an exception, branch misprediction or memory consistency violation as we will see below.

As in the multiple checkpointing mechanism for a uniprocessor, all the in-flight instructions remain speculative until their corresponding checkpoint commits. This means that memory instructions remain in the processor queues and do not modify the local cache or the global memory, and they are subject to a rollback. The oldest checkpoint in the processor can commit when all of its corresponding instructions, i.e. those that come after the checkpoint and before the next checkpoint, are finished. In Figure 3, for example, P_3 can commit checkpoint Chk_{31}, when all the instructions up to Chk_{32} have been completed. In the

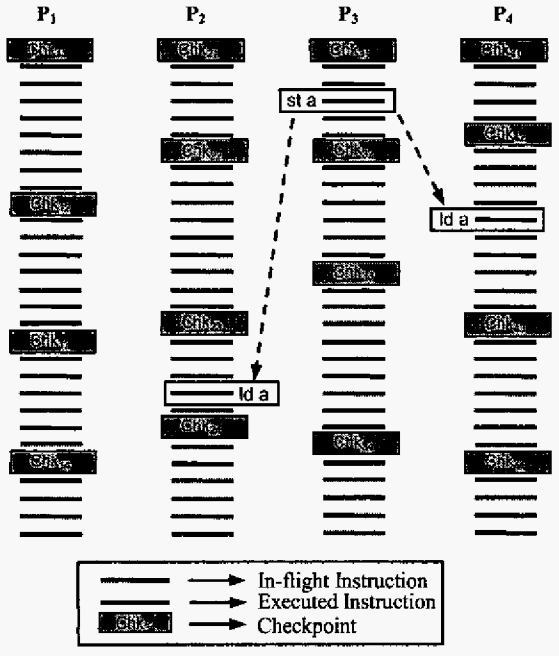

Figure 3: Execution flow for 4 processors.

KIMP system, this commit is followed by the atomic broadcast of all the cache tags that the processor has modified during the checkpoint execution, these are, the pending memory updates. By "atomic" we mean that after the processor gets access to the bus it does not release the bus until all the tags have been broadcast.

Remote processors snoop the memory updates searching for a conflict with the loads they have in their load queues, which are speculatively executed and are not already committed. In case of a conflict, the remote processor in question is forced to roll back, because it has speculatively used data that, at this point, is discovered to be "previously" modified. Thus, as long as conflicts are not found, speculatively executed instructions are not discarded. Finally, if no rollback happens, the speculatively executed instructions are globally performed when the checkpoint commits. In the example from Figure 3, the broadcasting of a store to a given memory location "a" conflicts with two other processors that have already speculatively loaded from location "a", but the loads have not been already committed. In this example, P_2 is rolled back to Chk_{23}, causing instructions from Chk_{24} to Chk_{23} to be discarded. Also P_4 roll back to Chk_{42}, forcing its newest instructions to be discarded.

This model makes the instructions between two checkpoints to behave as a single memory transaction, because they are executed speculatively and are globally and atomically performed when the corresponding checkpoint commits. Therefore, we call such groups of instructions *implicit transactions*, as they behave in a transactional manner, but they are not explicitly defined by the programmer or by any software level. In brief, our system acts like a transaction-based system, similar to the proposed TCC [14], but without any software support. This behavior allows the natural support for sequential consistency memory model.

Furthermore, multi-checkpointing yields improved performance if simple hardware modifications are made to support the concurrent execution of critical sections and speculative execution beyond synchronization points.

KIMP naturally allows concurrent execution of multiple threads in the same critical section, because the lock acquire is not performed until a checkpoint commits. However, a "silent stores elimination" [18] is needed to avoid unnecessary rollbacks due to the updating of the lock variable, when no actual data conflicts exist in the critical section. This mechanism only works when no checkpoint is taken inside a critical section. In addition, we can ensure that checkpoints are taken around a critical section by

detecting the lock entry and exit. Section 7 details these mechanisms.

Speculating beyond synchronization points can also be done, using a hardware mechanism to detect simple barrier constructs. Once a barrier is detected, a checkpoint is taken and the cache line corresponding to the flag variable is marked and changed to the expected value. This way the processor is able to execute the following instructions in a pure speculative mode, which means that no checkpoint can be committed. The speculated instructions can be rolled back if a memory conflict is found; otherwise they will wait for an update of the line associated with the flag variable, indicating that the expected and speculated value has been reached. Of course, any modification to the flag variable that does not correspond to the expected value will be ignored and will not produce a rollback. More details are given in section 8.

The recovery from violations for such speculative mechanisms is already included in Kilo-Instructions Processors, since they can already restore the system state from the previous checkpoint of an excepting instruction.

4. Implicit transactions

In the KIMP system, we define a *transaction* as the atomic execution of the instructions included between a committing checkpoint and the next one. The stores of a transaction are kept as a group, and are atomically released to the memory hierarchy only when the associated checkpoint is committed, updating the main memory and remote caches.

We say the proposed system executes *implicit transactions*. They are *implicit* because they are automatically hardware delimited, by means of the checkpointing mechanism, and the programmer does not need to know that the system provides such transactional behavior. Therefore, the ISA do not need any change, and current binaries can be directly executed. Note that this idea differs from the concept of transactions in previous works [14][23][1], where transactions are considered as programming constructs that normally need a hardware support.

4.1. Basic operation

As a processor speculatively executes the instructions within transactions, the read sets of the transactions (i.e. the memory locations referenced by load instructions) are stored in the processor load queue. In the same way, the write set (i.e. the store instructions) are temporarily kept in the store queues. In our snoopy bus based shared memory system, after a checkpoint commit, the stores are packaged and in-order broadcast over the bus, and the packet is snooped by the remote processors. This action globally validates all the speculative memory updates in a transaction, and with it, all the instructions in the transaction turn from speculative to executed.

During a transaction, the executed loads are speculative and, consequently, accessed memory locations must not change in order for the loads to remain valid. To verify this correctness, remote processors compare snooped modified addresses with the addresses of their write set in the processor load queues, and in case of a match, the processor rolls back to the checkpoint previous to the data use. In this case the remote processor has to re-execute instructions with new values. In absence of such a conflict, speculative instructions remain valid, and when all the instructions between two consecutive checkpoints finish, they can safely commit.

With such transaction-based system, forward progress of the parallel application is always guaranteed because rollbacks, other than normal exceptions or branch mispredictions, occur only when one processor is committing a transaction and the other ones have a conflict. Therefore, at least one processor, the one committing, always makes progress. The system does not care about conflicts that can exist during the execution of transactions because they are not revealed, i.e. not released to the memory hierarchy, until one transaction commits. Another related issue, is how to make the system fair, including avoidance of starvation; this is an issue partially covered by using adaptive transaction lengths, explained in next subsection.

Therefore, we have shown how KIMP provide a correctness substrate, constituted of speculative execution, atomic validation, and the rollback mechanism, which ensures that code is executed correctly.

4.2. Adaptive transaction length

The number of outstanding transactions and the number of instructions included in a single transaction directly depend on the maximum number of checkpoints and on the points where checkpoints are taken. Previous experience with a single Kilo-Instruction Processors dictates that a small number of checkpoints are enough.

In this multiprocessor transactional system, however, we propose that transaction lengths should be adaptive, decreasing in case of frequent rollbacks and increasing as long as no rollbacks occur. In case of frequent consistency violations, the length of the transactions decreases, also decreasing the number of violations and the number of instructions that are

discarded in case of a rollback. This is valid as our system, based on the correctness substrate indicated in the previous section, works properly independently of the instruction where a checkpoint is taken.

5. Memory Coherence

In KIMP, a snoopy bus based multiprocessor, the overall coherence protocol is simplified. Lines are not maintained in "shared" or "exclusive" state, as dictated by MESI-like cache coherence protocols. Our broadcast-based approach works as either a snoopy write invalidate or write update protocol as follows.

Basic Operation. During a transaction, the cache is not modified by local stores, only speculative loads are performed. Once a transaction commits, the pending stores are atomically performed and the broadcast mechanism transmits the changes to the rest of processors, updating or invalidating the remote cache lines. This way, when the stores from a transaction need to be broadcast, the corresponding processor will get control of the bus and release it only when all its memory updates are globally performed. This operation prevents other processor from broadcasting memory updates simultaneously and protects the memory system from coherence and consistency problems.

Finally, if there is an update or invalidation of a cache line, the corresponding processor will update or invalidate those cache lines matching a snooped address, and the execution will roll back to the previous checkpoint.

Broadcast Information. The contents of the broadcast message will determine the snoopy type: if the packet contains the written data, the protocol will behave as write update. Else, if the packet only contains the updated addresses, remote processors will invalidate those lines, and the protocol will work as write invalidate. Of course, it must be taken into account that using write update can put more pressure on the bus, because the data values have to be sent together with the modified addresses. In any case, if the processors running some specific parallel application need to send many stores when committing transactions, the resulting large broadcast packet can result in contention problems. However, we reduce the probability for such contention by intelligently merging and packaging stores as in [8].

When a processor needs to inform other processors that an address has been changed, the address sent to the bus is, in fact, the base address of the modified cache line. Therefore, in our system, the update or invalidate packet broadcast to the bus contains the set of the base addresses for the modified cache lines. The packaging mechanism we use, then, can reduce the number of addresses sent on a transaction commit, since several stores probably match the same cache line address as spatial locality indicates. This way, the number of addresses sent, for an update or invalidate packet, can be reduced up to a factor of n, where n is the number of memory locations that fit into a cache line. Furthermore, *silent store elimination* explained in section 7 can work together with the transaction packaging mechanism to effectively reduce the final number of memory addresses collected in a broadcast packet.

Scalability Problem. A snoopy bus allows atomic memory accesses because of the simultaneous broadcast capability of a bus and, therefore, the simple coherence mechanism we propose can be implemented in a straightforward manner. However, it is well known that buses do not scale well, so if we want to have the system to work with a larger number of processors we have to make the system use an alternative interconnection network. A directory-based KIMP using a packet-switched network is, therefore, an interesting alternative.

However, the directory-based approach lacks atomicity in the memory accesses due to the implicit unordered nature of packet-switched networks. This means that the implementation of the proposed coherence protocol under this model is no longer straightforward. The problem is that during the updating or invalidation of memory locations from a processor, other processors can simultaneously start their own updating or invalidation phases. This problem, then, would produce consistency problems due to the interleaving of updates from transactions of different processors.

In order to successfully implement a simple coherence protocol in a directory-based KIMP, an arbitration mechanism is needed. Arbitration would decide which processor can send its update packet to the rest of processors, avoiding the interleaving problem. Arbitration could be implemented, for instance, using a token-based mechanism. Briefly, the simplification of a normal coherence protocol, under this approach, would be the same as for a bus: no coherence state is needed for each memory address. However, it would be good for performance to maintain the list of sharers on each directory entry in order to reduce the number of messages.

6. Memory consistency

6.1. Consistency models

The memory consistency model of a shared-memory multiprocessor determines how the system can overlap or reorder memory operations. Different

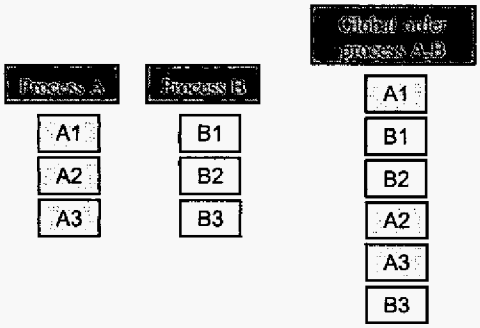

Figure 4: Sequentially consistent reordering of memory operations from 2 processors.

models offer a trade-off between programming simplicity and performance:

- Sequential Consistency (SC) [17], the most restrictive model, guarantees that interleaved memory operations from different processors appear to execute in program order, at the cost of a generally lower performance [28].
- On the other hand, less restrictive models, such as Release Consistency (RC) [11] provide a higher performance, at the cost of not ensuring strict ordering of memory operations.
- Other consistency models provide intermediate performance and restrictions, such as Processor Consistency [12].

Sequential Consistency is the most desirable model, as it is the simplest model to understand and it provides the most intuitive programming interface. A basic implementation of SC requires a processor to delay each memory access until the previous one is completed, what is simple but clearly leads to a low performance.

However, there are recent proposals that preserve the SC model without compromising performance. Some of these are:

- SC++ [13] makes use of hardware speculation for both load and store operations, and preserves SC by rolling back when a consistency violation is found. This way the system can rely on reordering and overlapping memory operations for performance similar to that achieved with the RC model.
- TCC [14] solves the problem of the consistency by proposing a parallel model based on software-delimited transactions, with a sequential ordering between them. Therefore, this model, which takes a software approach to the consistency problem, requires a new programming model to be used.

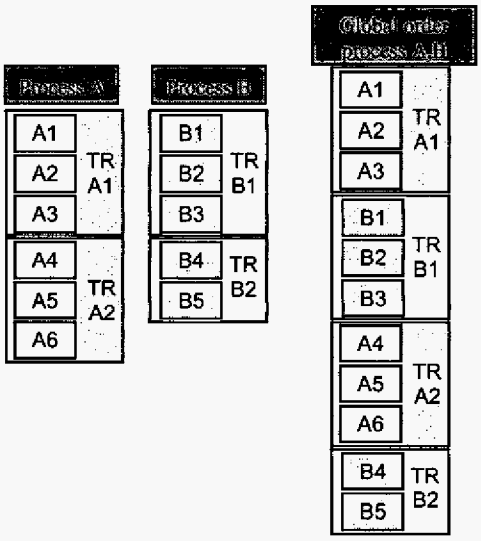

Figure 5: Sequentially consistent reordering of checkpoints from 2 processors.

6.2. KIMP maintains SC

Our proposal, based on the previously described transactional behavior, provides SC support in a natural manner: instead of assuring that single memory operations from a processor to be globally performed in order, we require full transactions to be in order. Fortunately, requiring an order for transactions inside a single processor is straightforward, because the checkpointing mechanism always commits the oldest transaction first.

Next, we explain with an example how we maintain SC. Sequential consistency requires that the result of any execution be the same as if the memory accesses executed by each processor were kept in order and with the accesses among different processors (arbitrarily) interleaved. In other words, we can have different global orders with different interleavings of memory operations from the different processors, but each of these interleavings must maintain all the individual program orderings. In Figure 4 we give an example of a sequentially consistent global ordering of memory operations from two different processors, labeled [A1, A2, A3] for processor A, and [B1, B2, B3] for processor B. The third column shows a global order that respects the program orders from processors A and B.

In the KIMP system we group memory accesses from each processor into *implicit transactions* by taking checkpoints. Thus, we can extend the definition of sequential consistency to such transactions, and require only transactions from each processor to be in order. The resulting global order will be an arbitrarily interleaved succession of transactions that will also

meet the basic definition of SC since it corresponds with one of the possible sequentially consistent global orderings. We show an example in Figure 5 where we group instructions into transactions, labeled [TR_A1, TR_A2] for processor A, and [TR_B1, TR_B2] for processor B. The third column shows that respecting the program order for those transactions will also respect program order for memory operations.

Furthermore, the KIMP environment allows an important improvement that avoids the latency that a simple SC implementation imposes: it allows the execution of memory operations out-of-order. KIMP allows this improvement without compromising the correctness of the SC model because:

1. The reordering and overlapping of memory operations is allowed only within a single transaction.
2. All the memory operations are executed speculatively and, while the loads bring data into the local cache, the stores are delayed until the transaction can commit without modifying any cache line.
3. During a transaction all the speculative loads that match the address of a previous pending store receive the correct value thanks to the usual store-forwarding mechanism.
4. The snooping of memory updates from the bus ensures that the values speculatively loaded are valid unless an address match is found, which would produce a rollback of the transaction.
5. Finally, the memory updates from the transaction are atomically broadcast only when the transaction commits, making the pending stores globally performed at this moment.

This way the global result of a transaction is the same, independently of the order of execution of its instructions, making the system behave as in figure 2.

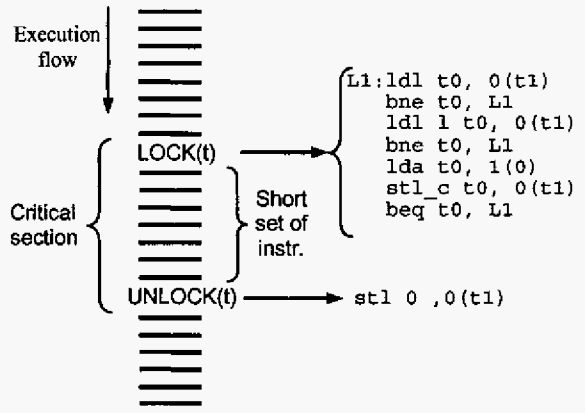

Figure 6: A typical critical section.

Therefore, in the figure an acceptable order for instructions in transaction "TR_A1", for example, could be "A2, A3 and A1", instead of the order shown: "A1, A2 and A3".

7. Locks

Lock structures control the access to critical sections by allowing only one process, the lock owner, to enter and to read and modify shared variables. A typical critical section, using load-linked and store-conditional instructions, is shown in Figure 6.

Critical sections ensure exclusive access to the lock owner, forcing any other threads desiring to enter the critical section to stall until the lock is released. Sometimes, this stall is unnecessary, as some threads may not actually modify any data, or they modify different fields of a shared data structure. In these cases, parallel execution could be allowed, avoiding the stall of threads waiting for the lock. Some examples, taken from [26] are shown in Figure 7.

The parallel execution of the critical section eliminates the dependence of other threads on the lock owner, which can sometimes generate very long waits, for example if the lock owner is suspended. The problem of critical sections is even more important in these cases.

7.1. Related work

In [26] Rajwar and Goodman show that some locks are too conservative and contention for shared data can occurs only under certain conditions. Thus, they propose SLE, a hardware approach that detects a typical Test&Test&Set lock construction and avoids acquiring it, leaving the critical section open and thus allowing several instances of the same critical section

```
a)      LOCK(locks->error_lock)
        if (local_error > multi->err_multi)
                multi->err_multi = local_err;
        UNLOCK(locks->error_lock)

b)      Thread 1

                LOCK(hash_tbl.lock)
                var = hash_tbl.lookup(X)
                if (!var)
                        hash_tbl.add(X);
                UNLOCK(hash_tbl.lock)

        Thread 2

                LOCK(hash_tbl.lock)
                var = hash_tbl.lookup(Y)
                if (!var)
                        hash_tbl.add(Y);
                UNLOCK(hash_tbl.lock)
```

Figure 7: Critical sections that, in most cases, admit parallel execution.

to execute concurrently. Correctness is preserved by detecting data collisions. All the data used during a speculative section is kept in the local cache, and a remote request to write that address forces the execution to roll back to the checkpoint. Thus, if no collision is detected, several instances of the same critical section can be executed in parallel, and if a collision happens, only one of them advances, forcing the other ones to restart with new data. This mechanism is improved in [27] by adding a timestamp to the write requests, which avoids process starvation, ensuring the forward progress of the oldest thread.

In [21] Martínez and Torrellas propose Speculative Synchronization. With specific lock instructions that help detecting the critical section entrance, they propose the execution of one safe thread, which actually acquires the lock, and multiple speculative threads, that detect the busy lock and execute speculatively. When the lock owner releases it, the rest of the threads commit their state, if they have already finished their critical section, or compete for ownership of the lock.

In [29] an improvement to the previous ideas is proposed. All the speculative threads execute their critical section, even though they may have a conflict. When all of the speculative threads can commit, an arbiter dictates the order in which they do so, minimizing collisions.

TCC [14], as stated in section 4, divides code into transactions which replace critical sections. Consequently, every access to shared data has to fall in the same transaction as the computation and the update of the data, and software locks are replaced with special instructions that mark a transaction change. This method naturally allows the parallel execution of critical sections, and detects conflicts in case of data collision, without the need of using control variables.

7.2. KIMP and critical sections

The correctness substrate composed of the transactional behavior and the collision detection mechanism ensures valid operation, including when executing critical sections. In this section, we study and propose some improvements that allow KIMP to execute critical sections in parallel, in those cases where there is no real data collision. As the code from a critical section is executed different checkpoints are taken. In Figure 8, cases a) to d) show different possible checkpointing schemes.

Basic KIMP behavior. When a processor reaches the lock, it checks the value of the lock and acquires it if it is free. This acquire operation remains in speculative state in the processor queues. If a new checkpoint is taken inside the critical section, the validation of the transaction that finishes inside the critical section will globally acquire the lock, forcing remote processors inside the critical section to rollback, due to the lock variable invalidation. After that, remote processors will not enter the critical section, due to the acquired lock. This case is shown in Figure 8 a), where a checkpoint is taken inside the critical section. The validation of transaction T1 will force any other thread executing inside the critical section to rollback, as the lock variable had been speculatively read. When the processor validates transaction T2, the critical section gets unlocked and remote processors can start executing it.

The same invalidation happens if no checkpoint is taken until unlocking the critical section. In this case, the lock variable is also written. Thus, the validation of a transaction that has entirely executed a critical section will cause any other processor that is executing it speculatively to rollback, which means that only a single valid processor stays inside a critical section. This case is shown in Figure 8 b), and it should be the most frequent case, due to the short nature of critical sections.

These examples show that basic KIMP ensures the correctness of code, wherever checkpoints are taken.

Enhancing KIMP. However, if the lock variable is the only interaction between different processors accessing the same critical section, parallel execution of the critical section is feasible. In case b), the store of the lock variable writes a value in a memory position that already contains that value, which constitutes a silent store. We propose the use of two mechanisms that allow KIMP to naturally speculate through critical sections.

Firstly, a *silent store detection and removal*

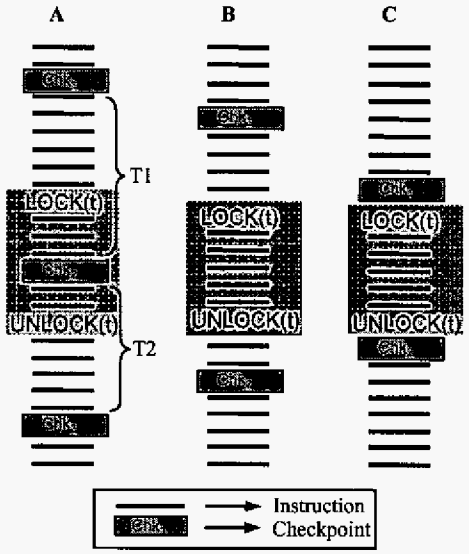

Figure 8: Different checkpointing schemes.

mechanism, which detects that the value does not globally modify memory, and removes that update from the update packet. When applied to a lock variable, the lock is removed from the update packet and this would allow a checkpoint configuration such as Figure 8 b) to be executed in parallel, avoiding processor stalls.

Additionally, we can ensure that no checkpoint is taken within a critical section. We make use of a *lock and barriers detection mechanism*, which dynamically detects typical Test&Test&Set lock constructs. To maximize parallelism, the lock detection hardware forces a new checkpoint to be taken just before the lock, so that the lock remains open for the rest of the processors at the beginning of the new transaction.

Normally, it is enough because critical section are short, as stated before, and the next checkpoint would fall after the unlock. However, the hardware could detect the lock release, which is a store to the lock variable, and take a new checkpoint just after it. This makes the transaction length equal to the critical section, and avoids unnecessary rollbacks due to collisions on data outside the critical section. This example is shown in Figure 8 c).

In case of multiple consecutive conflicts and rollbacks, the mechanism that adaptively changes the transaction length, would make processes to advance reducing transaction length as needed.

Finally, we note that the proposed method preserves correctness independently of the length of the critical section. TCC, which is a very similar transaction approach, locally buffers all the memory updates corresponding to a certain transaction. In case of a buffer overflow, the processor in TCC has to acquire the bus grant and not leave it up to the end of the transaction, thus blocking the rest of the system. KIMP in case of such an overflow, takes a new checkpoint and waits for the resources to free from previous checkpoints before continuing execution. Thus, in such case, the behavior would be similar to the one presented in Figure 8 a).

7.3. Silent stores and store merging

Recent work on value locality introduced the concept of *silent stores* [18]. A silent store is defined as a memory write that does not change the system state. In [18] it is shown that a non-trivial percentage of the stores are silent. There is other works that makes use of silent stores, but we will mention only speculative lock elision (SLE) [26] which is of interest to us. The SLE proposal tries to dynamically remove locks to allow critical sections to be executed concurrently. Nevertheless, in order to achieve lock removal, SLE needs to detect and elide the silent store pairs associated with the modification of the flag variable of the lock and unlock constructs of parallel applications. Therefore, SLE implements silent stores detection which is specifically designed for those silent store pairs of a critical section, without considering other possible silent stores.

In KIMP, we also need to remove these silent store pairs because, like SLE, we remove the lock from critical sections in order to concurrently execute them. However, the mechanism we propose does not just remove this specific type of silent store. We will remove all the possible silent stores that appear during a single transaction, which of course includes those silent stores associated with locks.

The mechanism we propose is quite simple -- an ordinary *store merging* mechanism reduces the number of stores to the same address within a transaction to only one, and *silent store removal* avoids broadcasting the remaining stores. Figure 9 shows an example of use of these mechanisms.

The *store merging* mechanism we use in KIMP is similar to that implemented in the Alpha 21164 [9]. Store merging consists of removing a store when a younger store is to the same address. If we apply this mechanism just before broadcasting a transaction, and before searching for silent stores, we do not have to care about race conditions as in [9]. The Alpha 21164 needs to ensure, for instance, that between these two stores there is no load to the same address. But in our system, at the point we perform the store merging, all the loads of the transaction are supposed to be executed. It is also true that our mechanism is simpler because, unlike the Alpha 21164, only stores of a single transaction are supervised.

After performing the *store merging*, we carry out the *silent store removal*. To detect silent stores we leverage the store-forwarding logic used in current

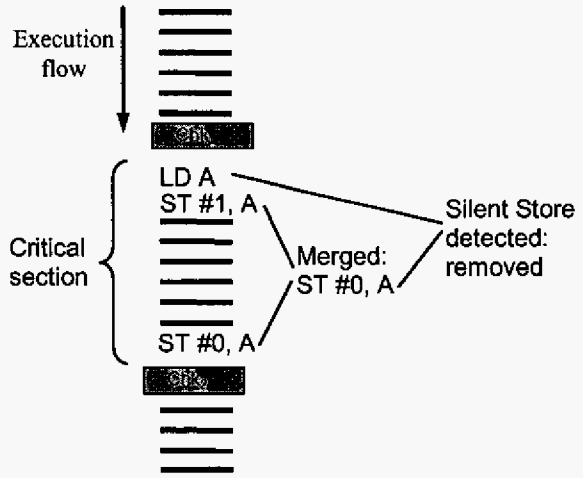

Figure 9: Silent store elimination.

processors. Store-forwarding searches the older stores before executing a load, and if an address match is found, the value of the store is forwarded to the load. Our mechanism, instead, searches older loads before executing a store, and if an address and value match is found, the store is removed because it is silent. Our silent store mechanism is simple because we just look for silent stores within a single transaction, and during a single transaction we have all the loads queued waiting to be committed and the stores queued waiting to be broadcast to the memory hierarchy.

8. Flags and barriers

Flags and barriers are used to synchronize different threads. A correct execution should make all threads wait for the barrier to open, before continuing execution. Similar to the previous case, this stall can be avoided in those cases, in which there is no real data interaction between different threads.

8.1. Related work

Speculative Synchronization [21] also deals with flags and barriers. When such a construct is detected, execution is continued further, in a speculative state. The instructions after a barrier remain speculative, waiting for the barrier to open before validating all the work.

TCC [14] substitutes flags and barriers by assigning phase number to each transaction. A processor can not commit a transaction if there is a remote pending transaction with a lower phase number. This way, transactions with the same phase numbers can commit in any order, whereas transactions with consecutive phase numbers are ensured to commit sequentially. Speculation is implemented naturally, as transactions after the phase change can be executed, but not committed.

8.2. KIMP mechanisms

KIMP can be adapted to speculate after barriers, in a similar manner to [21]. As in the previous section, there is the need to detect the barrier code and take a new checkpoint just prior to it. Speculative execution starts after the barrier, as shown in Figure 10. All the transactions after this barrier remain fully speculative, meaning that none of them can be validated, as long as the barrier remains closed. This is ensured by setting a "pure speculative mode" in the processor.

To determine the moment in which the barrier opens, the processor tracks the cache line containing the barrier variable, waiting for a cache event (an invalidation or an update of the line) to check the value again. When the barrier opens, the "pure speculative mode" is disabled, and the processor can start committing all the transactions in the pipeline. Note that a remote invalidation of the speculatively marked line does not force a rollback, but makes the speculative control unit check the variable again.

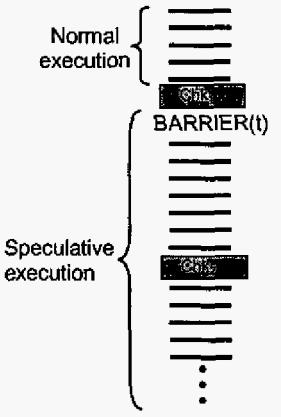

Figure 10: A speculated barrier.

Of course, all the speculative execution done before the opening of the barrier variable, has no effect on the consistency model. This is so, because the commit only happens after the barrier opening, and is the correctness substrate that ensures that the cache contents remain valid up to the commit instant.

The expected performance improvement of this scheme depends on the average time the processors wait at a barrier, while in the previous case the processor can commit the critical section and continue execution. Thus, if the waiting time does not exceed the time needed for the pipeline to fill and stall, we can achieve performance improvements. As Kilo-instruction Processors are designed to have thousands of in-flight instructions, this can occur frequently. Furthermore, in case of a conflict forcing a rollback, this mechanism will prefetch needed data, similar to [24], possibly reducing following memory latencies.

9. Conclusions

This paper introduces KIMP, a framework that makes Kilo-instruction Processors capable of executing parallel code in a transactional fashion, similar to the TCC model, but modifying neither the code nor the programming methodology. Our model maintains Sequential Consistency with a low hardware cost, a high performance potential and a reduced bus overhead. The hardware requirements are low, as most of the mechanisms are already proposed for kilo-instruction processors, and the processor model is simplified thanks to the transactional behavior.

Our model considers speculative execution in critical sections and barriers, reducing as much as possible the performance loss that these constructs cause in parallel programs. This, together with the advantages of transactional behavior, will provide high performance with no required code modifications.

Acknowledgements

This work has been supported by the Ministry of Education of Spain under contract TIN-2004-07739-C02-01 and grant AP2003-0539 (M. Galluzzi), the HiPEAC European Network of Excellence, and the Barcelona Supercomputing Center. J. E. Smith is partly supported by the NSF grant CCR-0311361. Per Stenström is partly supported by the Swedish Research Council under contract VR 2003-2576.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded Transactional Memory", In Proc. of the 11th HPCA, California, pp 316-327, Feb. 2005

[2] A. Cristal, M. Valero, J. Llosa, and A. González, "Large virtual ROBs by processor checkpointing", Tech. Rep. UPC-DAC-2002-39, UPC, Spain, July 2002

[3] A. Cristal, J. F. Martínez, J. Llosa, and M. Valero, "A Case for Resource-conscious Out-of-order Processors", In IEEE TCCA Comp. Architecture Letters, 2, October 2003.

[4] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Kilo-instruction processors", In Intl. Symp. on High Performance Computers, October 2003. LNCS 2858, 2003.

[5] A. Cristal, D. Ortega, J. Llosa, and M. Valero, "Out-of-Order Commit Processors", In Proc. of the 10th HPCA, February 2004.

[6] A. Cristal, O. Santana, M. Valero, J. F. Martínez, "Toward Kilo-instruction Processors", In ACM Transactions on Architecture and Code Optimization, V. 1, No. 4, Dec. 04.

[7] A. Cristal et al., "Kilo-instruction Processors: Overcoming the Memory Wall", To be published on IEEE Micro Magazine, Vol. 25, No. 3, May/June, 2005.

[8] F. Dahlgren, M. Dubois, and P. Stenström, "Combined Performance Gains of Simple Cache Protocol Extensions", in Proc. of 21st ISCA, pp. 187-197, April 1994.

[9] J. H. Edmondson, et al., "Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor", Digital Technical Journal, Vol. 7, No. 1, 1995, pp. 119--135.

[10] M. Galluzzi et al., "A First Glance at Kilo-Instruction based Multiprocessors", In Proc. of the 1st Conf. on Computing Frontiers, pp. 212-221, Ischia, Italy, April 2004.

[11] K. Gharachorloo et al., "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", In Proc. of the 17th ISCA, 1990.

[12] J. R. Goodman, "Cache Consistency and Sequential Consistency", Tech.Rep. no.61, SCI Committee, March 1989

[13] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is SC + ILP = RC?", In Proc. of the 26th ISCA, 1999

[14] L. Hammond et al., "Transactional Memory Coherence and Consistency", In Proc. of the 31st Annual International Symposium on Computer Architecture, Germany, June 2004.

[15] J. Huh et al., "Coherence Decoupling: Making Use of Incoherence", In Proc. of the 11th ASPLOS, October 2004

[16] T. Karkhanis and J.E. Smith, "A Day in the Life of a Data Cache Miss", In Proc. of the 2nd WMPI, 2002.

[17] L. Lamport, "How to make a Multiprocessor Computer that Correctly Executes Multiprocess Programs", IEEE Transactions on Computers, C-28(9):690-691, 1979.

[18] M. H. Lipasti, K. M. Lepak, "On the Value Locality of Store Instructions", In Proc. of the 27th ISCA, 2000

[19] M. M. K. Martin, M. D. Hill and D. A. Wood, "Token Coherence: Decoupling Performance and Correctness", In Proc. of the 30th ISCA, 2003

[20] J. F. Martinez, A. Cristal, M. Valero, and J. Llosa, "Ephemeral Registers", Technical Report CSL-TR-2003-1035, Cornell Computer Systems Lab, 2003.

[21] J. Martínez, J. Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Explicitily Parallel Applications", In Proc. of the 10th ASPLOS, Oct. 02

[22] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, and V. Vinals, "Delaying Physical Register Allocation Through Virtual-Physical Registers", In Proc. of the 32nd Intl. Symp. on Microarchitecture, pages 186–192, November 1999.

[23] K. E. Moore, M. D. Hill and D. A. Wood, "Thread-Level Transactional Memory", TR1524, Comp. Science Dept. UW Madison, March 31, 2005

[24] O. Mutlu et al., "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors", In Proc. 9th HPCA, pp. 129-140, 2003

[25] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton, "An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors", In Proc. Of the 7th ASPLOS, October 1996

[26] R. Rajwar, and J. R. Goodman, "Speculative Lock Elision: Enabling Highly-Concurrent Multithreaded Execution", In Proc. of 34th Intl. Symp. on Microarchitecture, pp.294-305, Dec. 2001.

[27] R. Rajwar, and J. R. Goodman, "Transactional Lock-Free Execution of Lock-Based Programs", In Proc. of the 10th ASPLOS, Oct. 02.

[28] P. Ranganathan, V.S.Pai, and S. Adve, "Using Speculative Retirement and Larger Instruction Window to Narrow the Performance Gap Between Memory Consistency Models", In Proc. of the 9th Symposium on Parallelism in Algorithms and Architectures. June, 1997.

[29] P. Rundberg, and P. Stenström, "Speculative Lock Reordering", In Proc. of IPDPS, April 2003.

[30] D. J. Sorin et al., "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery", In Proc. of the 29th ISCA, June 2002.